Algorithm: A Simpler Macro Processor

WILLIAM A. WARD, JR.
Computer Sciences Corporation

Macro processors have been in the computing tool chest since the late 1950s. Their use, though perhaps not what it was in the heyday of assembly language programming, is still widespread. In the past, producing a full-featured macro processor has required significant effort, similar to that required to implement the front-end to a compiler augmented by appropriate text substitution capabilities. The tool described here adopts a different approach. The text containing macro definitions and substitutions is, in a sense, ''compiled'' to produce a program, and this program must then be executed to produce the final output.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Program editors*; D.3.2 [**Programming Languages**]: Language Classifications—*Macro and assembly languages*; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*;

General Terms: Algorithms

Additional Key Words and Phrases: Awk, portable, simple

1. INTRODUCTION

Macro processors (MPs) have been recognized as useful tools since the late 1950s [Greenwald 1959]. Before the widespread use of high-level programming languages, MPs were indispensable aids provided by many assemblers [Kent 1969]. Later, MPs proved useful in the implementation of high-level programming languages themselves; the PL/I preprocessor [IBM 1981], the C preprocessor **cpp**, and **MAC-2** [Tomassini 1990] for Modula-2 are examples of this. Rational FORTRAN (Ratfor), a derivative of FORTRAN with additional features to facilitate structured programming, was implemented by using an MP as a preprocessor that allowed the use of if-else, for, and while control structures by defining them as macros [Ghezzi and Jazayeri 1987, p. 243]. The Algorithm 622 MP was developed to assist in the translation of the ELLPACK interface language into FORTRAN [Rice, Ribbens, and Ward 1984a]. [Rice, Ribbens, and Ward 1984b]

MPs may also play a significant role as tools to improve software developer productivity. Sometimes it is necessary to maintain multiple versions of the same program with minor variations between the versions; this is necessary when there are

Author's address: Computer Sciences Corporation, P.O. Box 820186, Vicksburg, MS 39182.

multiple hardware targets, when moving a code from one platform to another, and when the developer is considering alternative algorithms. This simple type of configuration management can generally be accomplished using an MP with a conditional statement. A recent example of this is seen in the use of Algorithm 622 to port geophysical software from a supercomputer to various UNIX workstations [Levin 1998]. In fact, the development of the MP described here was partially motivated by the need to perform this kind of version control on a numerical integration code.

In summary, MPs historically have been used both as software development tools and productivity aids and as important components for providing a higher level of abstraction in the langauge translation process. Further information on MPs may be found in Brown [1974] Campbell-Kelly [1973], and Cole [1976].

2. IMPLEMENTATION

2.1 Approach

MPs accept two types of source lines. *Text lines* are processed for macro substitution and are echoed to the output. *Directive lines*, or *directives*, on the other hand, actually constitute a programming language; they define macros and control program flow and are not echoed to the output. Providing for macro definition and substitution requires software for symbol table management, dynamic storage allocation, and string manipulation. Processing the command language requires a parser and interpreter, the complexity of which depends on the command language.

That a software version control problem motivated the development of this MP has already been noted. This problem had grown in complexity until it became apparent that neither **cpp** nor **m4** would suffice and that the best solution would be an MP whose command language was a fully-fledged programming language. Unfortunately, the effort required to produce such a tool from scratch would be similar to that for a compiler front-end plus an interpreter and would possibly be greater than that for the development project in which it would be used.

The solution to this dilemma was to define the directive language to be an existing language (awk) and then to use a two-pass approach. The first pass converts every input line to an awk statement, thus producing an awk program. In the second pass, the awk language processor executes the awk program to produce the final output. Appropriately, the program to implement the first pass is also written in awk. Such an approach was proposed by Brown [1974, pp. 77-79] who suggested PL/I as the embedded directive language. The simpler MP described here was christened **m5** (**m4**++).

2.2 Why Use Awk?

There are several scripting languages that could have been used to implement **m5**, including Perl and Python. In fact, it is easy to see how one language could have been

used for the translator, and another for the embedded directive language. However, this author believes that awk has some real advantages, as noted below (these are admittedly opinions, so take them with a grain of salt).

- Convenience. **awk** was bundled with the locally installed operating system; obviously there are other freely available scripting languages, but they were not installed, and would have had to be fetched from a remote site and compiled. If **m5** is to be installed on any UNIX system, there is a high probability that **awk** is already installed.

- Suitability. Awk provides many features useful to the MP developer and user, including a concatenation operator, associative arrays, dynamic storage management for variable length strings, dynamic type conversion, and a number of useful built-in string handling functions.

- Simplicity. Awk arguably has a simpler syntax than, say, Perl and it is a smaller, more compact language.

- Learnability. Partially as a consequence of 2., and partially because its syntax is strongly reminiscent of C, yet without some of C's annoyances (semicolons and declarations, for instance, are really not crucial to a MP language).

3. USAGE

3.1 Overview

The program that performs the first pass noted above is called the *m5 translator* and is named **m5.awk**. The input to the translator may be either standard input or one or more files listed on the command line. An input line with the directive prefix character (# by default) in column 1 is treated as a directive statement in the MP directive language (awk). All other input lines are processed as text lines. Simple macros are created using awk assignment statements and their values referenced using the substitution prefix character ($ by default). The backslash (\) is the escape character; its presence forces the next character to literally appear in the output. This is most useful when forcing the appearance of the directive prefix character, the substitution prefix character, and the escape character itself.

3.2 Macro Substitution

All input lines are scanned for macro references that are indicated by the substitution prefix character. Assuming the default value of that character, macro references may be of the form $var, $(var), $(expr), $[str], $var[expr], or $func(args). These are replaced by an awk variable, awk variable, awk expression, awk array reference to the special array `M[]`, regular awk array reference, or awk function call, respectively. These are, in effect, macros. The MP translator checks for proper nesting of parentheses and double quotes when translating $(expr) and $func(args) macros, and checks for proper

nesting of square brackets and double quotes when translating $[expr] and $var[expr] macros. The substitution prefix character indicates a a macro reference unless it is (i) escaped (e.g., `\$abc`), (ii) followed by a character other than A-Z, a-z, (, or [(e.g., `$@`), or (iii) inside a macro reference (e.g., `$($abc)`; probably an error).

An understanding of the implementation of macro substitution will help in its proper usage. When a text line is encountered, it is scanned for macros, embedded in an awk print statement, and copied to the output program. For example, the input line

```
The quick $fox jumped over the lazy $dog.
```

is transformed into

```
print "The quick " fox " jumped over the lazy " dog "."
```

Obviously the use of this transformation technique relies completely on the presence of the awk concatenation operator (one or more blanks).

3.3 Macros Containing Macros

As already noted, a macro reference inside another macro reference will not result in substitution and will probably cause an awk execution-time error. Furthermore, a substitution prefix character in the substituted string is also generally not significant because the substitution prefix character is detected at translation time, and macro values are assigned at execution time. However, macro references of the form $[expr] provide a simple nested referencing capability. For example, if `$[abc]` is in a text line, or in a directive line and not on the left hand side of an assignment statement, it is replaced by `eval(M["abc"])`/. When the output program is executed, the **m5** runtime routine **eval()**/ substitutes the value of `M["abc"]` examining it for further macro references of the form $[str] (where "str" denotes an arbitrary string). If one is found, substitution and scanning proceed recursively. Function type macro references may result in references to other macros, thus providing an additional form of nested referencing.

3.4 Directive Lines

Except for the include directive, when a directive line is detected, the directive prefix is removed, the line is scanned for macros, and then the line is copied to the output program (as distinct from the final output). Any valid awk construct, including the function statement, is allowed in a directive line. Further information on writing awk programs may be found in Aho, Kernighan, and Weinberger [1988], Dougherty and Robbins [1997], and Robbins [1997].

3.5 Include Directive

A single non-awk directive has been provided: the include directive. Assuming that # is the directive prefix, #include(filename) directs the MP translator to immediately read

from the indicated file, processing lines from it in the normal manner. This processing mode makes the include directive the only type of directive to take effect at translation time. Nested includes are allowed. Include directives must appear on a line by themselves. More elaborate types of file processing may be directly programmed using appropriate awk statements in the input file.

3.6 Main Program and Functions

The MP translator builds the resulting awk program in one of two ways, depending on the form of the first input line. If that line begins with "function", it is assumed that the user is providing one or more functions, including the function "main" required by **m5**. If the first line does not begin with "function", then the entire input file is translated into awk statements that are placed inside "main". If some input lines are inside functions, and others are not, awk will will detect this and complain. The MP by design has little awareness of the syntax of directive lines (awk statements), and as a consequence syntax errors in directive lines are not detected until the output program is executed.

3.7 Output

Finally, unless the -c (compile only) option is specified on the command line, the output program is executed to produce the final output (directed by default to standard output). The version of **awk** specified in ARGV[0] (a built-in awk variable containing the command name) is used to execute the program. If ARGV[0] is null, **awk** is used.

4. EXAMPLE

Understanding this example requires recognition that macro substitution is a two-step process: (i) the input text is translated into an output awk program, and (ii) the awk program is executed to produce the final output with the macro substitutions actually accomplished. The examples below illustrate this process. # and $ are assumed to be the directive and substitution prefix characters. This example was successfully executed using **awk** on a Cray C90 running UNICOS 10.0.0.3, **gawk** on a Gateway E-3200 runing SuSE Linux Version 6.0, and **nawk** on a Sun Ultra 2 Model 2200 running Solaris 2.5.1.

4.1 Input Text

```
#function main() {

    Example 1: Simple Substitution
    ------------------------------
#   br = "brown"
    The quick $br fox.

    Example 2: Substitution inside a String
```

```
    ---------------------------------------
#   r = "row"
    The quick b$(r)n fox.

    Example 3: Expression Substitution
    ----------------------------------
#   a = 4
#   b = 3
    The quick $(2*a + b) foxes.

    Example 4: Macros References inside a Macro
    -------------------------------------------
#   $[fox] = "\$[q] \$[b] \$[f]"
#   $[q] = "quick"
#   $[b] = "brown"
#   $[f] = "fox"
    The $[fox].

    Example 5: Array Reference Substitution
    ---------------------------------------
#   x[7] = "brown"
#   b = 3
    The quick $x[2*b+1] fox.

    Example 6: Function Reference Substitution
    ------------------------------------------
    The quick $color(1,2) fox.

    Example 7: Substitution of Special Characters
    ---------------------------------------------
\#   The \$ quick \\ brown $# fox. $$
#}
#include(testincl.m5)
```

4.2 Included File testincl.m5

```
#function color(i,j) {
    The lazy dog.
#   if (i == j)
#       return "blue"
#   else
#       return "brown"
#}
```

4.3 Output Program

```
function main() {
    print
    print "    Example 1: Simple Substitution"
    print "    ------------------------------"
    br = "brown"
    print "    The quick " br " fox."
    print
    print "    Example 2: Substitution inside a String"
    print "    ---------------------------------------"
    r = "row"
    print "    The quick b" r "n fox."
    print
    print "    Example 3: Expression Substitution"
    print "    ----------------------------------"
    a = 4
    b = 3
    print "    The quick " 2*a + b " foxes."
    print
    print "    Example 4: Macros References inside a Macro"
    print "    -------------------------------------------"
    M["fox"] = "$[q] $[b] $[f]"
    M["q"] = "quick"
    M["b"] = "brown"
    M["f"] = "fox"
    print "    The " eval(M["fox"]) "."
    print
    print "    Example 5: Array Reference Substitution"
    print "    ---------------------------------------"
    x[7] = "brown"
    b = 3
    print "    The quick " x[2*b+1] " fox."
    print
    print "    Example 6: Function Reference Substitution"
    print "    ------------------------------------------"
    print "    The quick " color(1,2) " fox."
    print
    print "    Example 7: Substitution of Special Characters"
    print "    ---------------------------------------------"
    print "\#  The \$ quick \\ brown $# fox. $$"
}
function color(i,j) {
    print "    The lazy dog."
    if (i == j)
        return "blue"
    else
```

```
        return "brown"
}

function eval(inp ,isplb,irb,out,name) {

    splb = SP "["
    out = ""

    while( isplb = index(inp, splb) ) {
        irb = index(inp, "]")
        if ( irb == 0 ) {
            out = out substr(inp,1,isplb+1)
            inp = substr( inp, isplb+2 )
        } else {
            name = substr( inp, isplb+2, irb-isplb-2 )
            sub( /^ +/, "", name )
            sub( / +$/, "", name )
            out = out substr(inp,1,isplb-1) eval(M[name])
            inp = substr( inp, irb+1 )
        }
    }

    out = out inp

    return out
}
BEGIN {
    SP = "$"
    main()
    exit
}
```

4.4 Final Output

```
    Example 1: Simple Substitution
    ------------------------------
    The quick brown fox.

    Example 2: Substitution inside a String
    ---------------------------------------
    The quick brown fox.

    Example 3: Expression Substitution
    ----------------------------------
    The quick 11 foxes.
```

```
    Example 4: Macros References inside a Macro
    -------------------------------------------
    The quick brown fox.

    Example 5: Array Reference Substitution
    ---------------------------------------
    The quick brown fox.

    Example 6: Function Reference Substitution
    ------------------------------------------
    The lazy dog.
    The quick brown fox.

    Example 7: Substitution of Special Characters
    ---------------------------------------------
#   The $ quick \ brown $# fox. $$
```

5. CONCLUDING REMARKS

The implementation approach just described has several advantages:

- The implementation is short and simple because a fully-fledged parser is not required. The relative brevity of the code inherently reduces code complexity and improves maintainability.

- The use of an existing and thoroughly tested language processor (**awk**) greatly enhances **m5**'s reliability.

- In contrast to **m4**, **cpp**, Algorithm 622, and many other MPs, **m5** contains a complete high-level language (awk), including conditionals, loops, and functions with arguments.

- Awk is widely used and is similar to C; this enhances **m5**'s accessibility.

- Awk is distributed with the UNIX operating system as a standard utility. There are at least three versions of **awk** for which source code is freely available, and at least two commercial versions. Versions exist for MS-DOS, OS/2, Windows 95, and Windows NT [Dougherty and Robbins 1997, pp. 255-277]. This wide availability promotes the portability of **m5**.

Several possible enhancements to **m5** are under consideration, including improvement of the dynamic macro substitution capability, translation to C to improve performance, and reimplementation in Perl and Python to enhance the directive language capabilities.

ACKNOWLEDGEMENTS

The author gladly recognizes the inspiration provided for this work by Dr. John R. Rice of Purdue University and dedicates it to him on the occasion of his 65th birthday.

The development and initial testing of this algorithm was performed on equipment donated by Sun Microsystems under Academic Equipment Grant #EDUD-NAFO-970211. The author gratefully acknowledges Sun's generosity.

This paper is published in the interest of scientific and technical information interchange; the ideas and findings contained herein should not be construed as an official position of the U.S. Army. Use of any trademarks in this study is not intended in any way to infringe on the rights of the trademark holder.

REFERENCES

AHO, ALFRED V., KERNIGHAN, BRIAN W., AND WEINBERGER, PETER J. 1988. *The AWK Programming Language*, Addison-Wesley.

BROWN, P. J. 1974. *Macro Processors and Techniques for Portable Software*, John Wiley & Sons.

CAMPBELL-KELLY, M. 1973. *An Introduction to Macros*, Macdonald.

COLE, A. J. 1976. *Macro Processors*, Cambridge University Press.

DOUGHERTY, DALE AND ROBBINS, ARNOLD 1997. *sed & awk*, O'Reilly & Associates.

GHEZZI, CARLO AND JAZAYERI, MEHDI 1987. *Programming Language Concepts*, John Wiley & Sons.

GREENWALD, I. 1959. A technique for handling macro instructions, *Comm. ACM*, **2/11 (Nov.)**, 21-22.

IBM 1981. OS and DOS PL/I Language Reference Manual, GC26-3977-0, IBM Programming Publishing.

KENT, W. 1969. Assembler language macro processing: a tutorial oriented towards the IBM 360, *Computing Surveys*, **1/4 (Dec.)**, 183-196.

LEVIN, STEWART A. 1998. Remark on algorithm 622: a simple macro processor, *ACM Trans. Math. Softw.*, **24/3 (Sep.)**, 336-340.

RICE, J. R., RIBBENS, C. J., AND WARD, W. A. 1984. The template processor, In *Solving Elliptic Problems Using ELLPACK*, by John R. Rice and Ronald F. Boisvert, ed., Springer-Verlag, 469-484.

RICE, JOHN R., RIBBENS, CALVIN, AND WARD, WILLIAM A. 1984. Algorithm 622: A simple macro processor, *ACM Trans. Math. Softw.*, **10/4 (Dec.)**, 410-416.

ROBBINS, ARNOLD 1997. *Effective AWK Programming*, Specialized System Consultants.

TOMASSINI, MARCO 1990. MAC2—a macro processor for Modula-2, *J. of Pascal, Ada & Modula-2*, **9/1 (Jan.)**, 28.